Improving time-to-insight with data delivery and columnar analysis

K. Choi[A], B. Galewsky[B], P. Onyisi[A], G. Watts[C], M. Weinberg[D] (and many others)

A. University of Texas at Austin, B. University of Illinois at Urbana-Champaign, C. University of Washington
D. University of Chicago

Snowmass Computational Frontier Workshop

8/11/2020

Classic analysis vs columnar analysis

Classic workflow: xAOD or miniAOD → flat ntuples → skimmed ntuples → histograms → plots

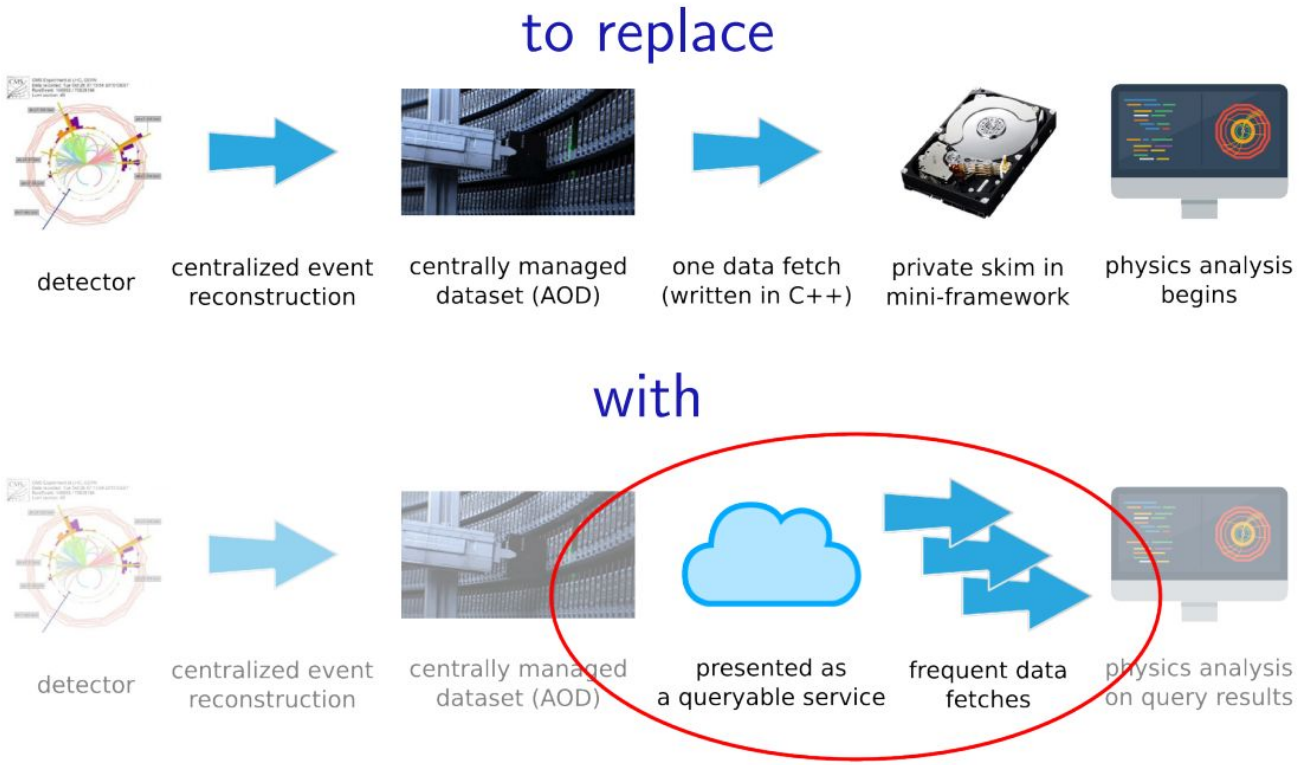
- Starting formats are prescribed, but enormous variation after that

Standard analysis might have “primary ntuple”

- Write/maintain specialized ntuplization code
- Submit jobs to HTCondor by hand
- Skim/trim; some data replicated (many times)
- Selections/cutflows baked into analysis
- Can’t add new variables

Columnar workflow: recast data into contiguous columns instead of by event

- Much more efficient for processing
- Add columns to existing data
- Cache only data necessary for computation



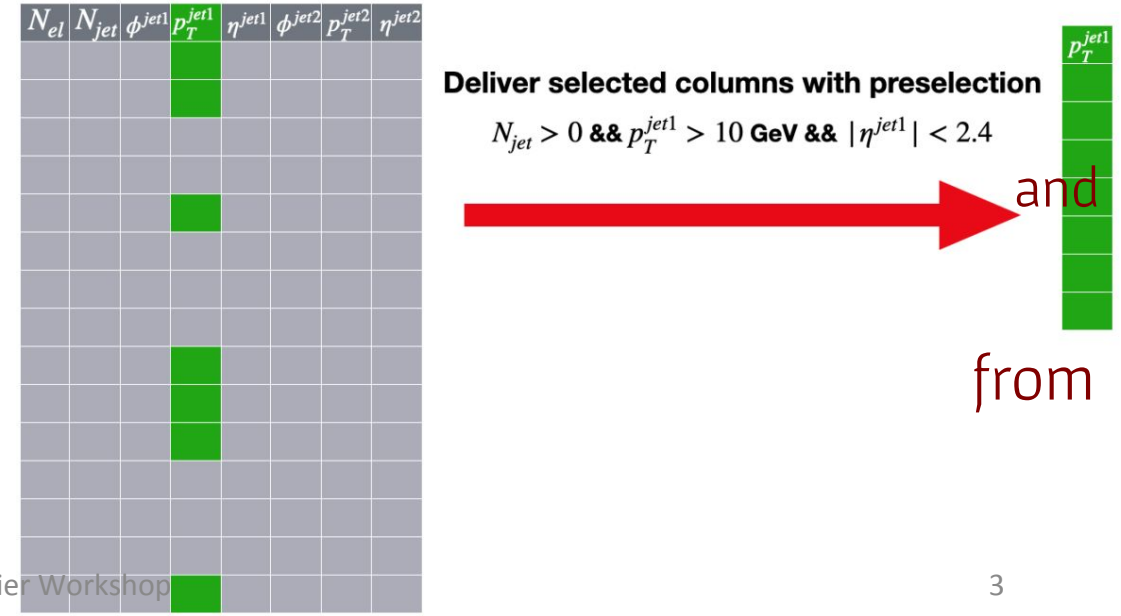
ServiceX: a columnar data delivery service

Experiment-agnostic service to enable on-demand data delivery from grid to analyzers.

Tailored for **nearly-interactive,** high-performance **array-based analyses**

- Provide uniform interface to data storage services; users don't need to know how or where data is stored
- Capable of **on-the-fly data transformations** into variety of formats (flat ROOT files, Arrow buffers, Parquet files, Pandas dataframes, …)
- Pre-processing functionality: Unpack compressed formats, filter events in place, compute new variables, project data columns

Designed specifically for columnar analyses. Start any format, extract only needed columns



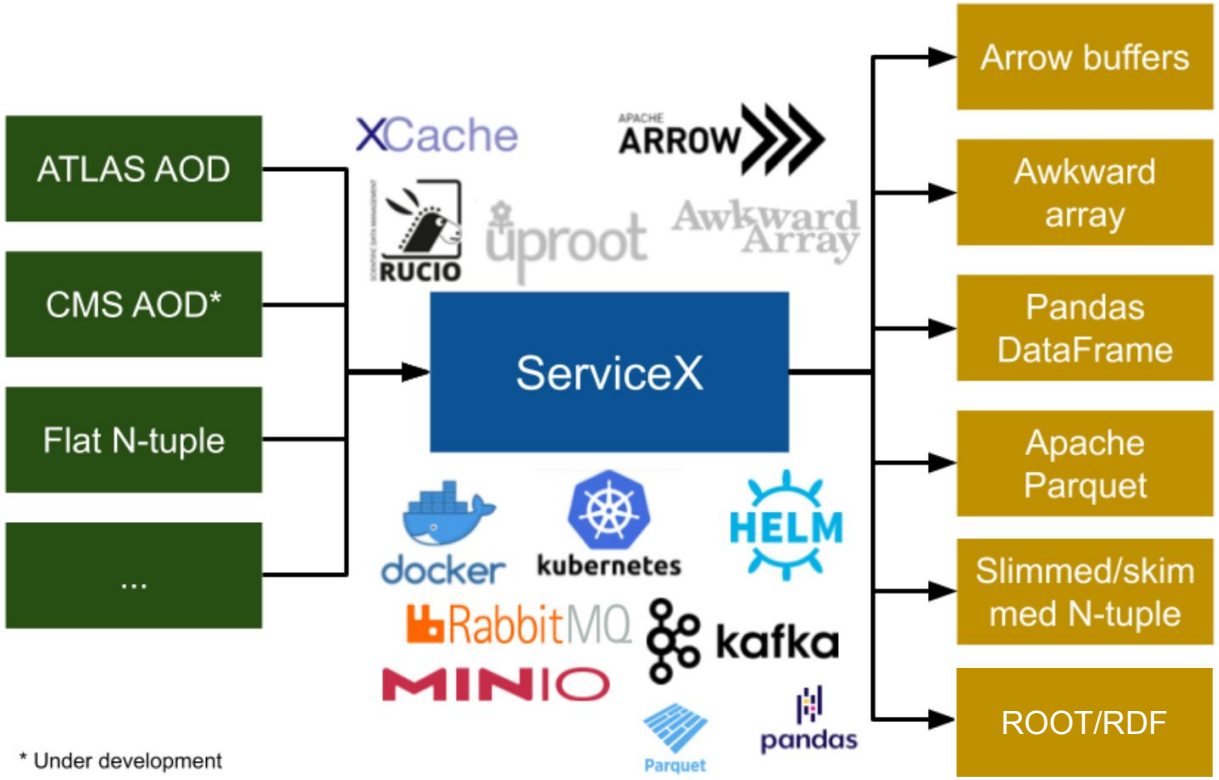
ServiceX outline

Users specify events, columns, output format, filters

- ServiceX associates request with unique ID
- Queries Rucio on behalf of user
- Can optionally attach XCache for input caching
- Validates request and performs data transformation
- Output storage available via MinIO object store or Kafka message broker

ServiceX under the hood

- System orchestrated via Kubernetes: self-healing transformer pods, adaptive based on resources
- Helm chart for easy deployment: supports scaling to other clusters, including Tier 2s



Current work and future directions

Have functional transformers for multiple formats across ATLAS and CMS experiments

Moving toward v1.0 release with full documentation and client libraries

Lots of extensions based on community interest!

- Fast, convenient, cached access to skimmed/trimmed data from flat formats (PHYSLite, nanoAOD)
- Explore connections other columnar tools
- Short term: can adapt service to existing workflows
  - Users bring ntuplizers, we “transformerize” them to use service
- Longer term: scaling to multiple highly configurable instances
  - Support scaling to multiple ServiceX deployments across clusters, including Tier 2s
  - Users bring only a config; use existing, maintained transformers
  - Plan to replace ntuplization, free groups from development/maintenance of ntuplizer code
  - Augment flat formats with custom upstream columns (from xAOD, miniAOD, RECO, …)

Backup slides

Delivering data to new analysis platforms

First introduced in Feb 2018 whitepaper: delivery of data from lakes to clients (insulate from upstream systems, remove latencies, reformat, filter & possibly accelerate)

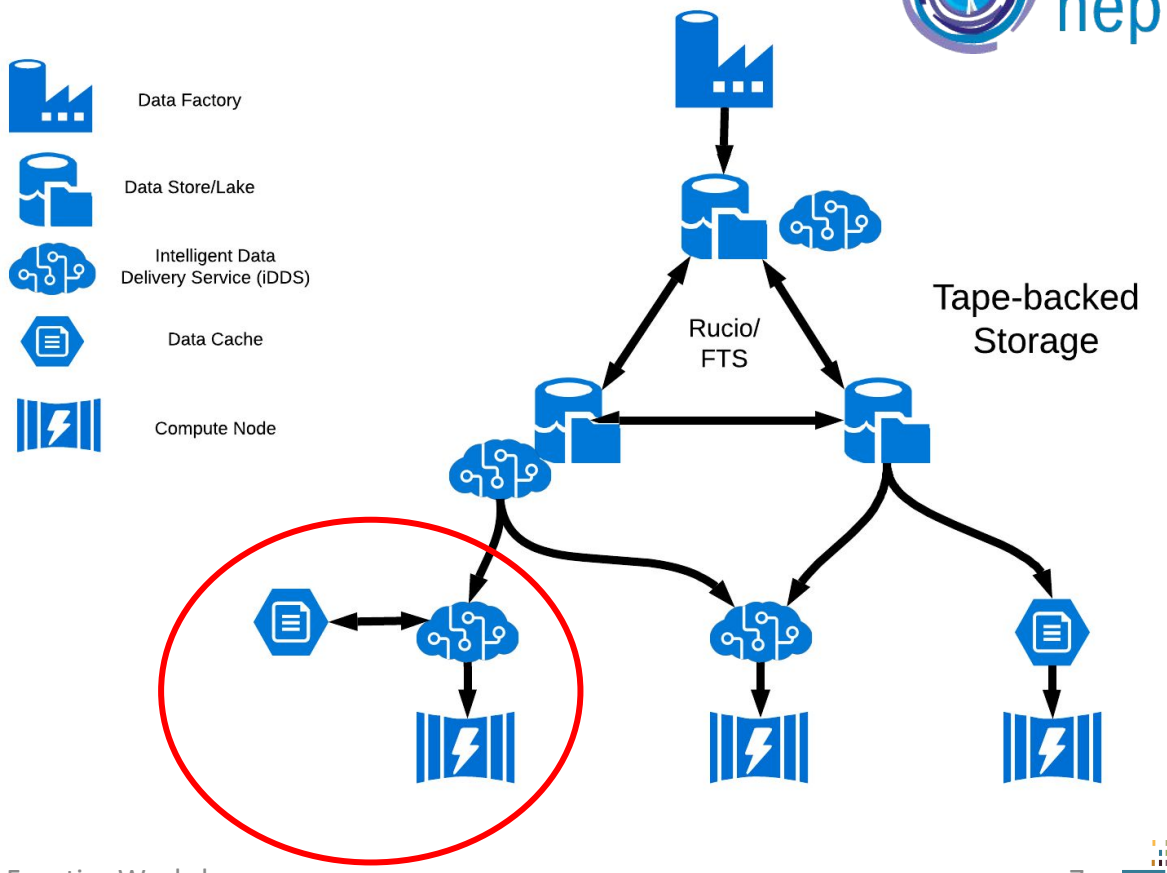
ServiceX focuses on integration with Rucio and reformatting for pythonic tools & **endstage analysis systems**

DOMA/AS groups interested in ***R&D*** for data delivery for analysis of columnar and other data formats

Supports multiple input types (xAOD, flat ntuples, …) and common data mgt (Rucio, XCache)

Utilize industry standard tools (GKE, on-prem Kubernetes, Helm, Kafka, Redis, Spark, …)

Reproducible, portable deployments



IRIS-HEP R&D efforts in DOMA & Analysis Systems

- DOMA/AS groups interested in ***R&D*** for data delivery for analysis of columnar and other data formats
- Supports multiple input types (xAOD, flat ntuples, …) and common data mgt (Rucio, XCache)
- Utilize industry standard tools (GKE, on-prem Kubernetes, Helm, Kafka, Redis, Spark, …)
- Reproducible, portable deployments

Marc Weinberg
University of Chicago

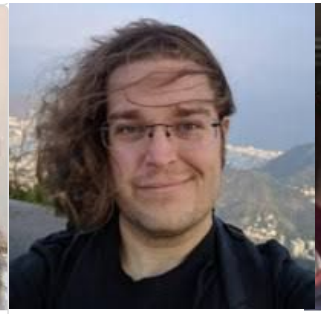
Ben Galwesky
National Center for Supercomputing Applications

Mark Neubauer
University of Illinois at Urbana-Champaign

Gordon Watts
University of Washington

Ilija Vukotic

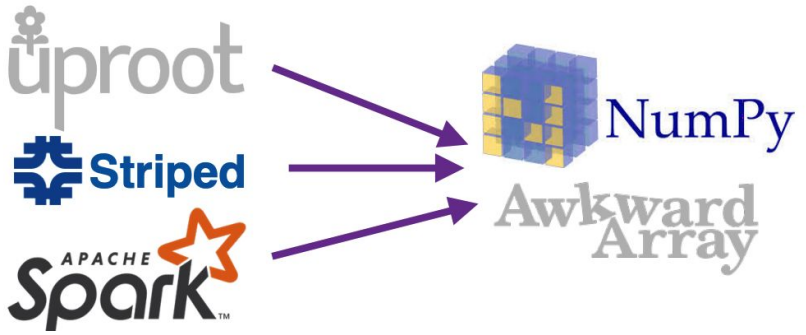
Jim Pivarski
Princeton University



Lindsey Gray
Fermilab

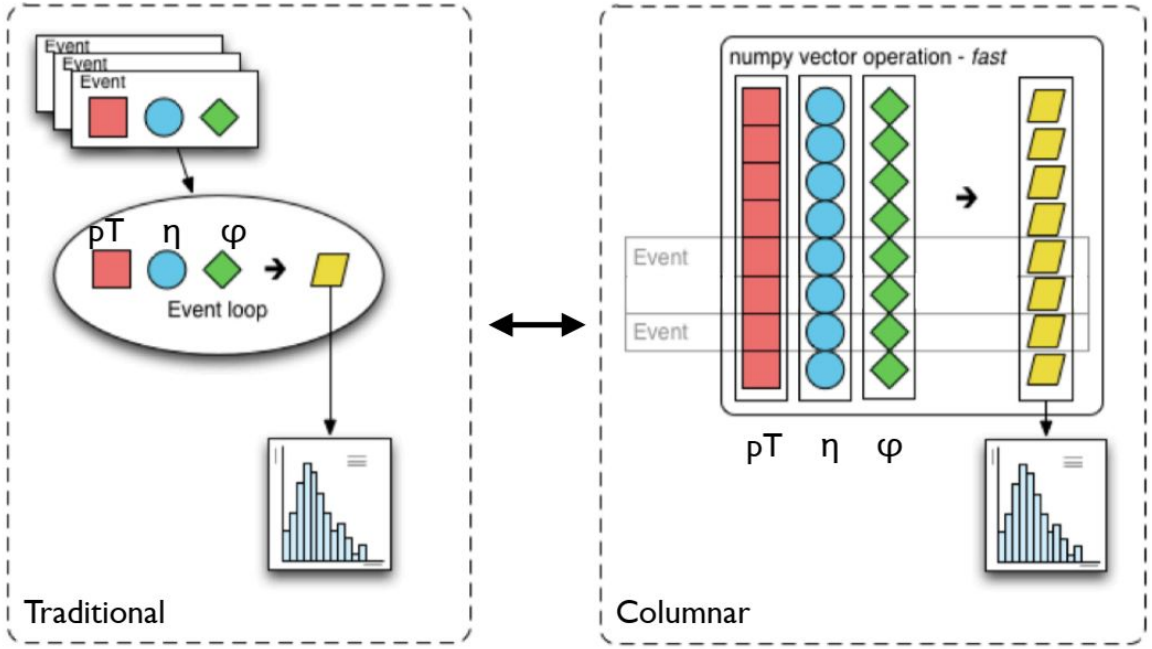
Rob Gardner
University of Chicago

Columnar data R&D efforts



Recast data so attributes of physics objects grouped into contiguous columns, rather than grouping by event and then object

- Much more efficient for processing!
- Updating event content (or corrections) can be done by adding columns to existing data
- Can cache only necessary data for computation;
  No longer need to load entire event in memory



However, this is a significant change for analyzer

- New syntax can be simpler, more expressive
- Imagine analysis code with no for() loops…

Loop-less array programming

- But this is shown to the user as a list containing lists of various lengths:

```
In [4]: import uproot
        f = uproot.open("HZZ-objects.root")
        t = f["events"]

In [5]: a = t.array("muoniso")      # muon isolation variable; multiple per event
        a
```

Event 1 Event 2 ...

```
Out[5]: <JaggedArray [[4.2001534 2.1510613] [2.1880474] [1.4128217 3.3835042] ... [3.7629452], [0.550810
        7], [0.]] at 7b229f313240>
```

The implementation is a façade: these are not millions of list objects in memory but two arrays with methods to make them *behave like* nested lists.

```
In [6]: a.offsets
Out[6]: array([   0,    2,    3, ..., 3823, 3824, 3825])

In [7]: a.content
Out[7]: array([4.2001534, 2.1510613, 2.1880474, ..., 3.7629452, 0.5508107, 0.       ], dtype=float32)
```


Loop-less array programming

Can do all kinds of stuff with optimized linear algebra computations

- Multidimensional slices of data
- Element-wise operations (e.g. `muons_pz = muons_pt * sinh(muons_eta)`)
- Broadcasting (e.g. `muon_phi - 2 * pi`)
- Event masking, indexing, array reduction, etc.

But we don't have a simple rectangular arrays

- Nested variable-size data structures everywhere in HEP
- Jagged arrays handle this with two 1D arrays:
  - First array contains long list of values, one per object
  - Second array contains breaks that give event boundaries

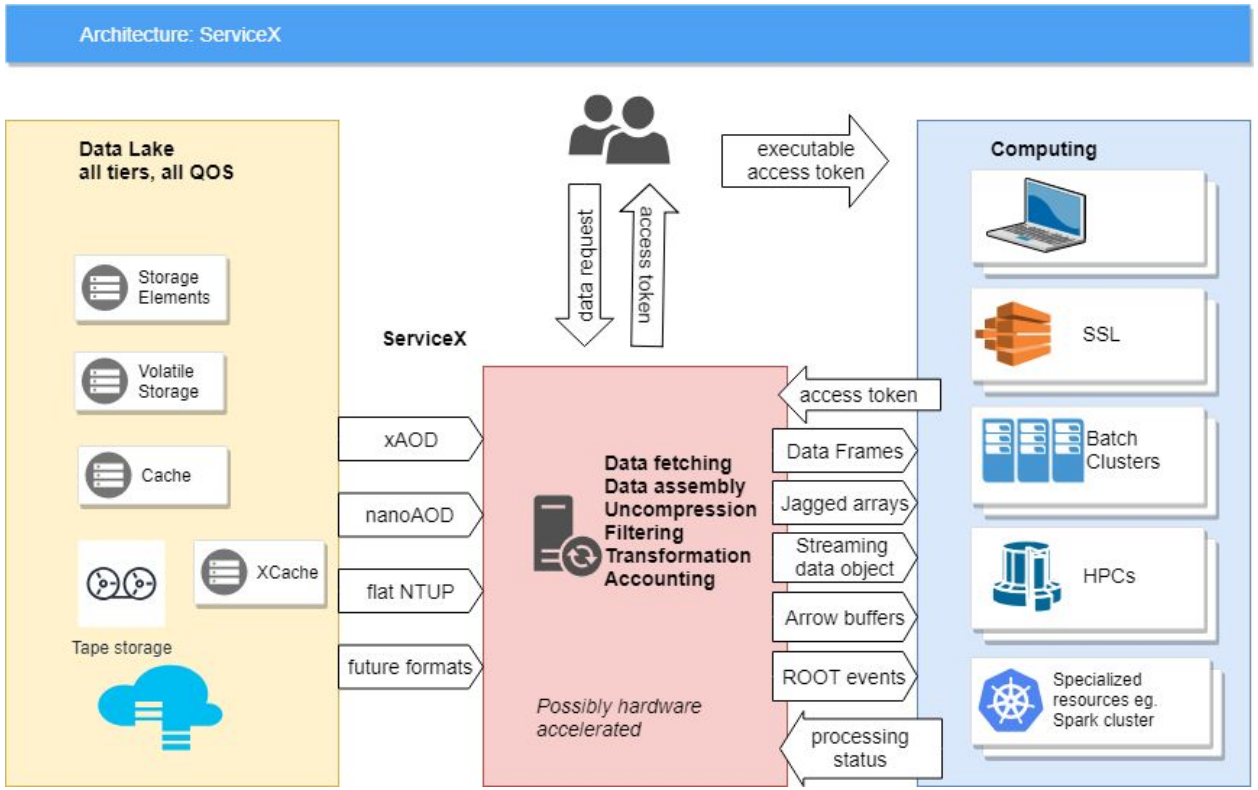
ServiceX components

Users specify needed events/columns and desired output format

- Use metadata tags (real/sim data, year, energy, run number, …)
- Any required preselection

ServiceX

- Queries backend (Rucio) to find data
- Gives unique token to identify request
- Access data from storage (optionally cached via XCache)
- Validates request and extract requested columns, perform data transformations
- Send output to object store or message broker for analysis



ServiceX implementation

System designed to be modular

- Can switch out modules to transform different types of input data, swap schedulers, …

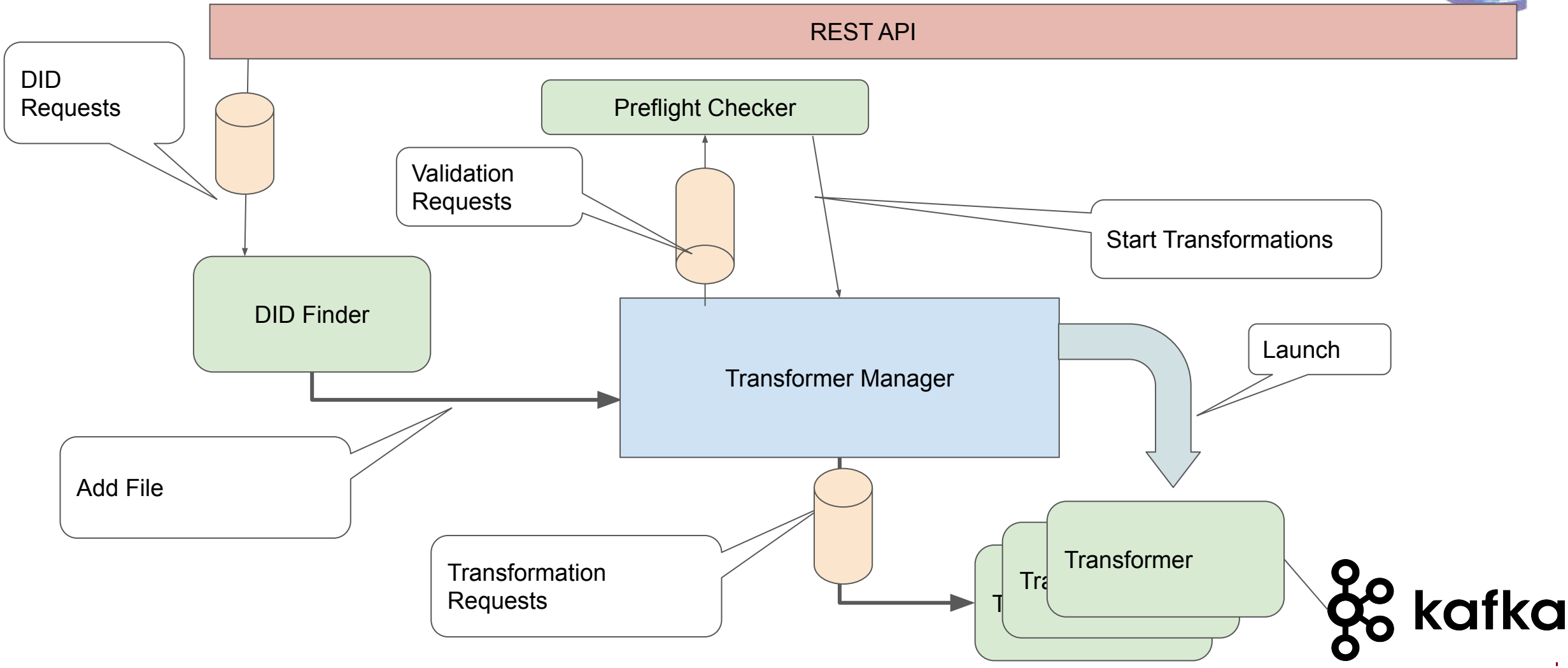
Implemented as central service in Kubernetes cluster on Scalable Systems Lab (SSL) cluster

- Easy to deploy: Just use Helm chart to define plugins to run
- Service can be deployed on variety of systems, including individual laptops
- **Reproducible pattern** for deployment on Kubernetes clusters (e.g. **Tier2s**, **institutional k8s T3**?)

Composed of multiple deployments: REST API server, DID finder, transformer, message broker

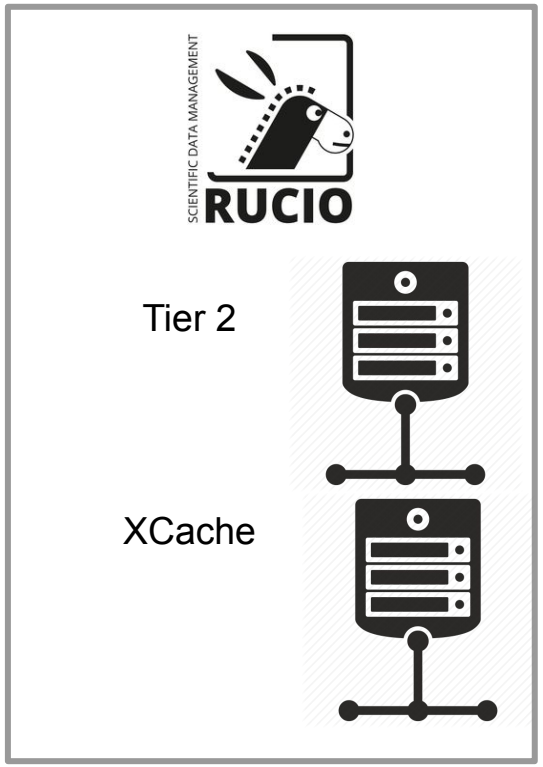
- API server: Manages requests via RabbitMQ with Postgres DB
- DID finder: Queries data lake via Rucio, writes ROOT files to XCache
- Transformer: Takes input files from XCache, outputs in various formats (ROOT files with flat trees, Awkward arrays, Parquet, Arrow tables, …)
- Kafka manager: Receives input from producer (transformer) and makes topics available to consumers (analysis jobs)

ServiceX architecture

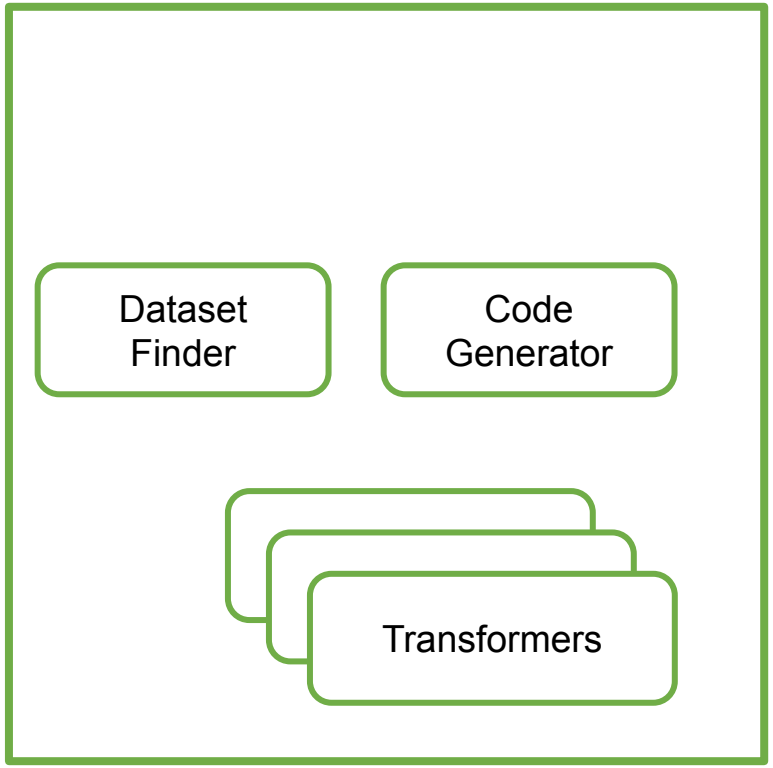


ServiceX in the IRIS-HEP ecosystem

Data Lake



ServiceX

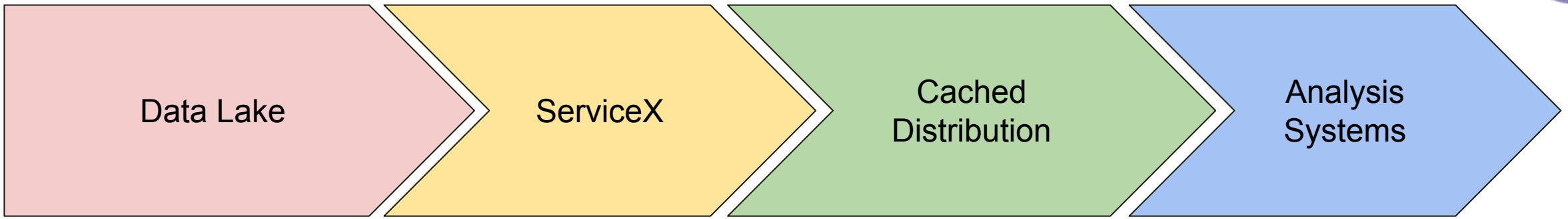


Cached Distribution

Analysis Facility

IRIS-HEP Scalable Systems Lab

Connections to DOMA



ServiceX is part of DOMA's iDDS

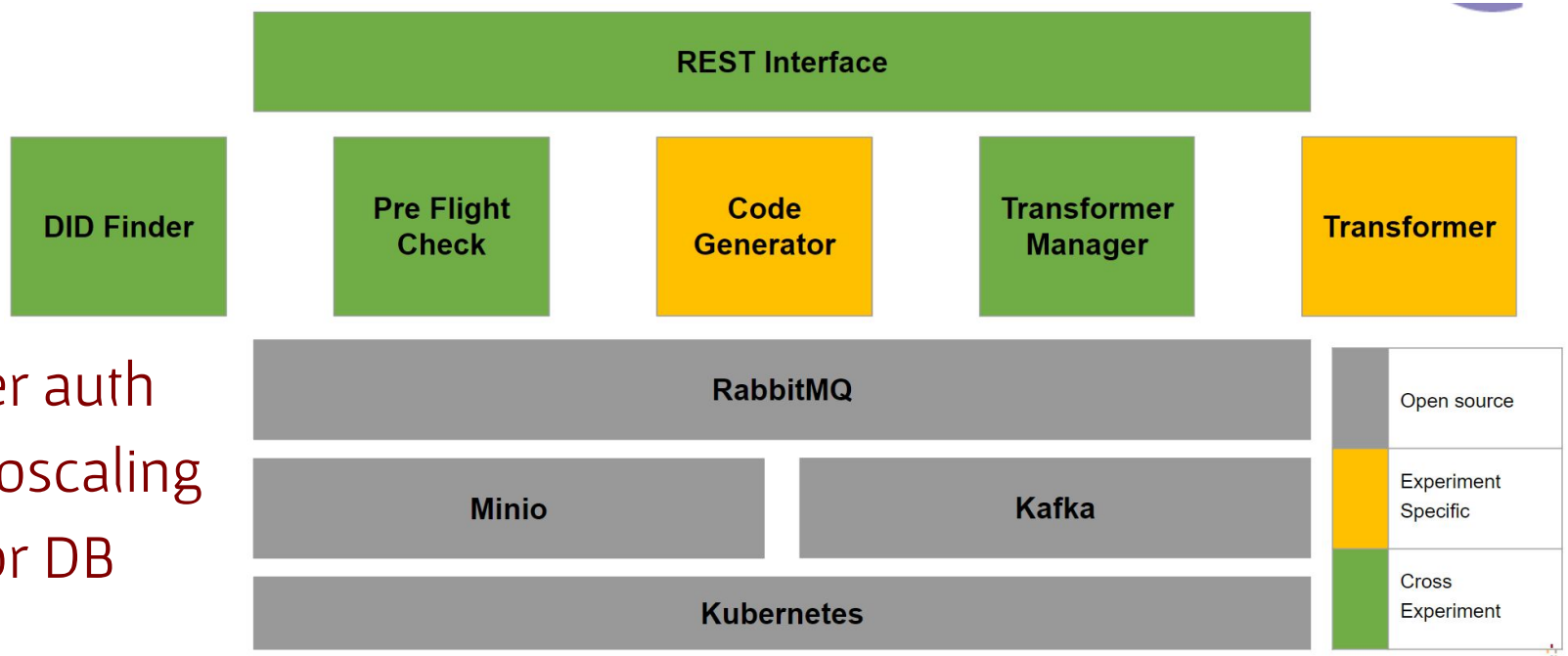
- feeds data to downstream analysis systems
- enables data transformations developed in individual environments to be scaled up to production-based operations

ServiceX is being prototyped using IRIS-HEP's Scalable Systems Lab

- includes reproducible pattern for deployment
- entire project implemented as central service in Kubernetes cluster on SSL
- takes advantage of SSL infrastructure support to develop new features quickly.

ServiceX so far

- Have functional transformers for multiple formats across multiple experiments
  - xAOD/DAOD inputs for ATLAS and miniAOD inputs for CMS
  - Flat TTrees and nanoAOD inputs via Uproot tools
- Moving toward ServiceX v1.0 release, new bells and whistles
  - Splash page, docs
  - Client "frontend" libraries for easy user interaction
  - Admin tools, new user auth
  - Configurable pod autoscaling
  - Persistent volumes for DB



ServiceX running on SSL: RIVER 2

Services

```
NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
release-name-0-external             NodePort    10.104.248.66   <none>        19092:31090/TCP
release-name-1-external             NodePort    10.102.224.207  <none>        19092:31091/TCP
release-name-kafka                  ClusterIP   10.110.20.190   <none>        9092/TCP
release-name-kafka-exporter         ClusterIP   10.97.197.166   <none>        9308/TCP
release-name-kafka-headless         ClusterIP   None            <none>        9092/TCP
release-name-minio                  ClusterIP   10.97.66.31     <none>        9000/TCP
release-name-postgresql             ClusterIP   10.96.46.224    <none>        5432/TCP
release-name-postgresql-headless    ClusterIP   None            <none>        5432/TCP
release-name-rabbitmq               ClusterIP   10.106.225.45   <none>        4369/TCP,5672/TCP,25672/TCP,15672/TCP,9090/TCP
release-name-rabbitmq-headless      ClusterIP   None            <none>        4369/TCP,5672/TCP,25672/TCP,15672/TCP
release-name-servicex-app           NodePort    10.102.57.95    <none>        8000:31973/TCP
release-name-zookeeper              ClusterIP   10.108.18.68    <none>        2181/TCP
release-name-zookeeper-headless     ClusterIP   None            <none>        2181/TCP,3888/TCP,2888/TCP
```

- Currently deployed on Kubernetes cluster on SSL RIVER 2
- Performance testing: Running with hundreds of transformers on cluster
- Has Prometheus monitoring dashboard

Workloads

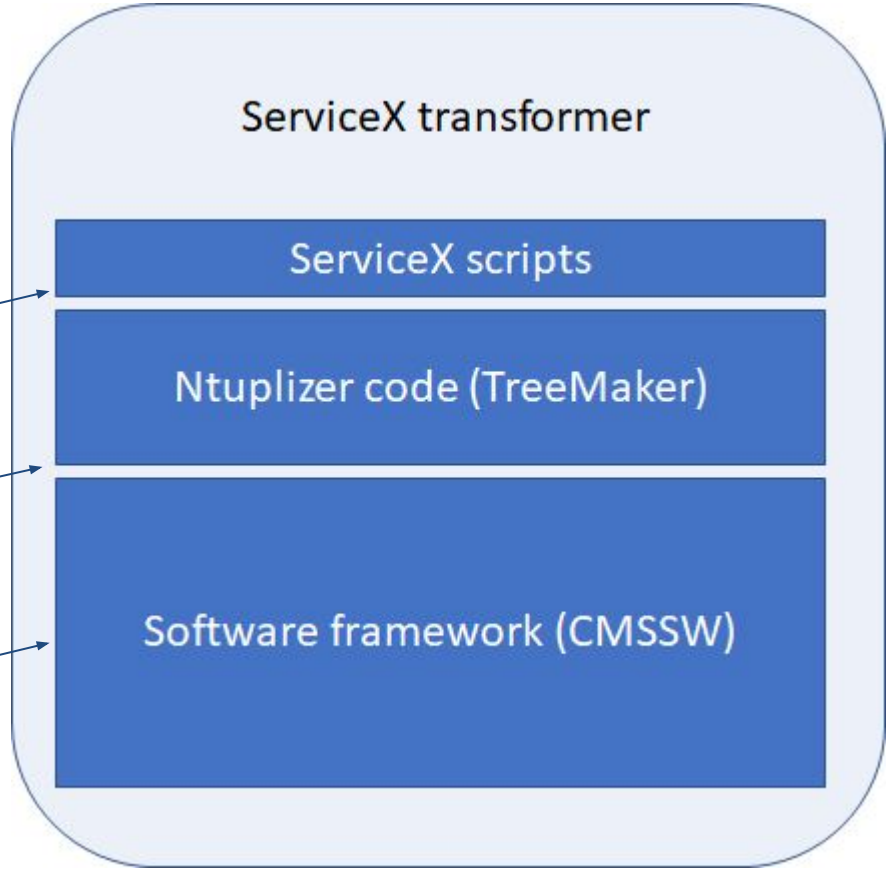
```
C:\Users\ivukotic>kubectl get pods -n servicex
NAME                                                READY   STATUS      RESTARTS   AGE
release-name-did-finder-b5b6bfb48-d2bnq             1/1     Running     2          14m
release-name-kafka-0                                1/1     Running     3          17d
release-name-kafka-1                                1/1     Running     1          17d
release-name-kafka-exporter-76cf9c7c69-nbzcq        1/1     Running     2          17d
release-name-minio-5c54b8648-b92n2                  1/1     Running     0          14m
release-name-postgresql-0                           1/1     Running     0          14m
release-name-preflight-cb454d56-f4tmv               0/1     Error       7          14m
release-name-rabbitmq-0                             2/2     Running     0          14m
release-name-servicex-app-9b58465fc-q8xkm           1/1     Running     2          14m
release-name-test-topic-consumer                    0/1     Completed   0          2d23h
release-name-test-topic-consumer-v2                 0/1     Completed   0          2d23h
release-name-test-topic-consumer-v3                 0/1     Completed   0          2d23h
release-name-test-topic-create-consume-produce      0/1     Completed   0          17d
release-name-testclient                             0/1     Error       0          26h
release-name-zookeeper-0                            1/1     Running     1          17d
release-name-zookeeper-1                            1/1     Running     2          17d
release-name-zookeeper-2                            1/1     Running     1          17d
transformer-210b2510-0e16-4006-9055-d9addb51fcaa-xprdk   1/1     Running     0          13m
```


ServiceX performance on ATLAS data

- 10TB across 7794 files (~ 1.3GB/file)
- 10-column test: reads ~ 10% of file
  - Time for transformer to process file: ~ 13 seconds
    - Single-transformer rate: 27.4 MB/s
  - Output size ~ 3MB (~ 400 reduction from input)
  - 400-transformer test completes in < 10 minutes
- 100-column test: reads ~ 30% of file
  - Total time to process file: ~ 31 seconds
    - Single-transformer rate: 11.5 MB/s
  - Output size ~ 38MB (~ 35 reduction from input)
  - 400-transformer test completes in < 25 minutes

Building a CMS transformer

- Set up CMS-specific deployment on RIVER with all the new stuff
  - Deployed by Alexx in separate namespace
- The transformer essentially built of three sets of layers
  - Thin top layer with a couple of scripts to talk to ServiceX and run the ntuplizer (basically one script to run things and one to grab grid proxy)
  - Layer of ntuplization code; currently using TreeMaker from Kevin and Alexx as a basis
  - Base layer for SW framework (needs to be containerized once/version/experiment)



Incorporating the ServiceX frontend libraries

- Installing client library

```
python -m pip install servicex==2.0.0b9
python -m pip install func-adl-xAOD==1.1.0b4
```

- Creating a request framework

```
import servicex
dataset = 'mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.e3601_s25
sx_endpoint = 'http://rc1-xaod-servicex.uc.ssl-hep.org'
minio_endpoint = 'rc1-xaod-minio.uc.ssl-hep.org'
ds = servicex.ServiceXDataset(
    dataset,
    servicex.ServiceXAdaptor(sx_endpoint, username='mweinberg', password='XXXXXXXXX'),
    servicex.MinioAdaptor(minio_endpoint)
    )
```

- Using helper functions to make a complex query
  - Make filter and compute new variable

```
r = f_ds \
    .Where('lambda e: e.Jets("AntiKt4EMTopoJets") \
        .Where('lambda j: j.pt() / 1000.0 > 30.0').Count() >= 1') \
    .Select('lambda e: e.Electrons("Electrons")') \
    .Select('lambda e: e.Select(lambda ele: ele.eta() * ele.phi())') \
    .AsAwkwardArray('EleMyVar') \
    .value()
```


ServiceX demo

Sample transform request including dataset to be transformed and output columns

Development version; some of these decisions will be hidden from the user

User receives unique request ID

Rudimentary updates on the progress of the transformation

To be augmented with status plots

ServiceX demo

This notebook illustrates the use of ServiceX to create a request for specific columns of data from a file stored in Rucio, and the reading of these columns to create analysis plots.

In [1]:
```
import requests
import tempfile
import pyarrow.parquet as pq
import pyarrow as pa
import awkward
from confluent_kafka import Consumer, KafkaException
import uproot_methods
from coffea import hist
import matplotlib.pyplot as plt
```

Transform the data

We start with the creation of the request inside the service. The user specifies the dataset to be transformed, along with the columns of interest. Here we transform a 700GB MC dataset of ~ 2 million Z -> ee events distributed across 17 files. The columns from this dataset are then streamed to Kafka, a message broker which makes them available to the user for analysis.

Meanwhile, the service returns a unique string that serves as the ID of the request. The user can use the request ID to get updates on the progress of the system and to identify their data in the message broker.

In [2]:
```
response = requests.post("http://release-name-servicex-app.servicex:8000/servicex/transformation", json={
    "did": "mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.e3601_s2576_s2132_r6630_r6264_p2363
    "columns": "Electrons.pt(), Electrons.eta(), Electrons.phi(), Electrons.e()",
    "image": "sslhep/servicex-transformer:v0.1-prep",
    "result-destination": "kafka",
    "kafka":{
        "broker": "release-name-kafka:9092"
    },
    "chunk-size": 3000,
    "workers": 17
})

print("Request ID:", response.json()['request_id'])
request_id = response.json()["request_id"]
status_endpoint = servicex_endpoint+"/transformation/{}/status".format(request_id)
```

```
Request ID: 826ea2b2-0849-4790-9427-afe6498348eb
```

Get updates on the transformation

Once the request is sent, we can get information back on the status of the request. Note that the transformer begins running as soon as the first files from the dataset are found within Rucio; some of the information does not become available until all the files are discovered (e.g. the total number of files remaining).

In [9]:
```
status = requests.get(status_endpoint).json()

print("Request ID:                ", status['request-id'])
print("Number of files processed:  ", status['files-processed'])
print("Number of files remaining:  ", status['files-remaining'])
if status['stats']:
    print("Number of events processed: ", status['stats']['total-events'])
    print("Size of files processed:    ", status['stats']['total-bytes'] / 1.0e9, "GB")
    print("Number of batches sent:     ", status['stats']['total-messages'])
    print("Total time:                 ", status['stats']['total-time'] / (n_workers * 60), "min")
```

```
Request ID:                826ea2b2-0849-4790-9427-afe6498348eb
Number of files processed:  17
Number of files remaining:  0
Number of events processed: 1993800
Size of files processed:    0.2005268 GB
Number of batches sent:     666
Total time:                 1.2245098039215687 min
```

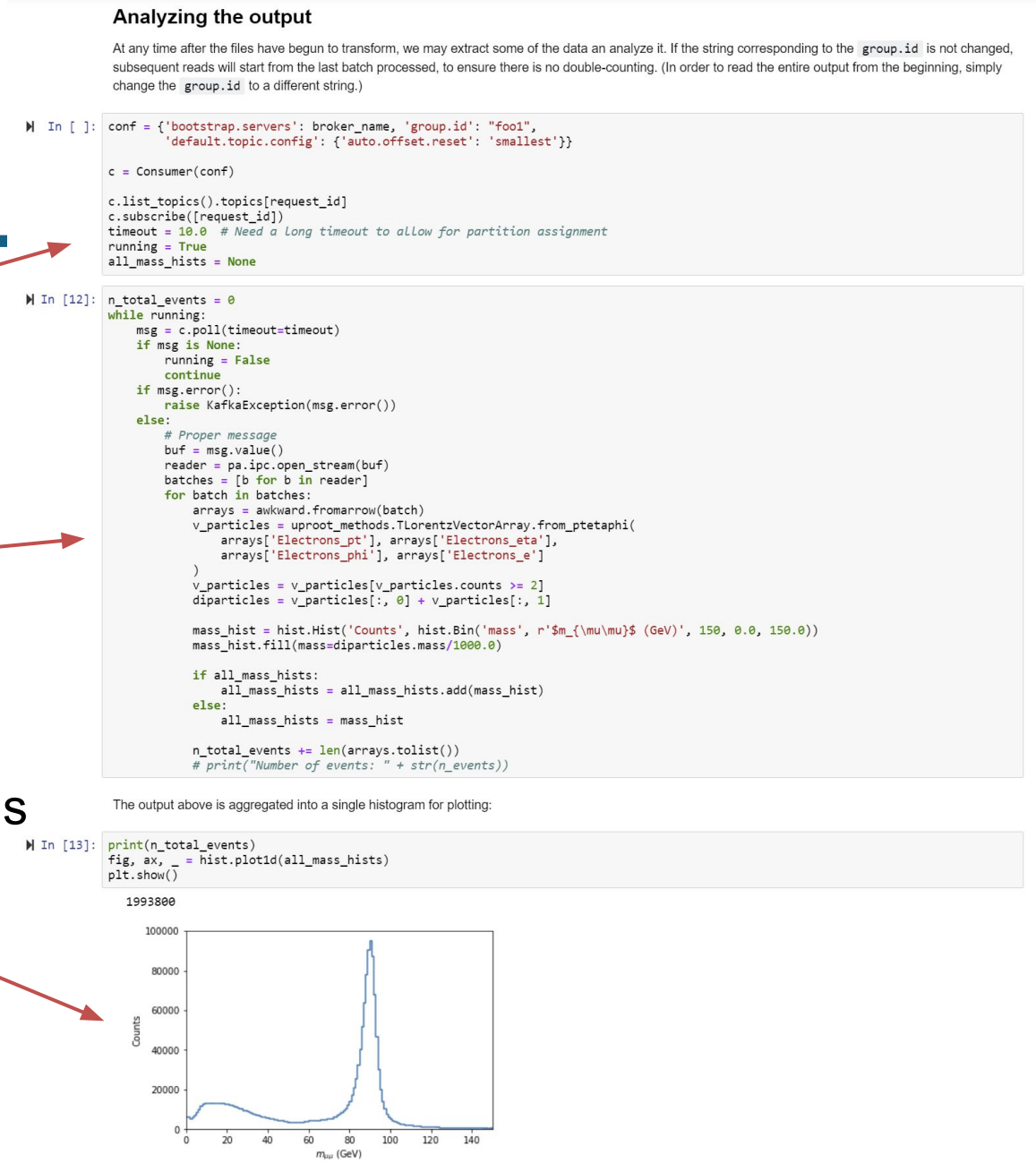
ServiceX demo

Output cached in message broker

Can be read out asynchronously, re-run

User analysis code

Time to go from xAOD or DAOD to analysis plot only a couple of minutes



Analyzing the output

At any time after the files have begun to transform, we may extract some of the data an analyze it. If the string corresponding to the `group.id` is not changed, subsequent reads will start from the last batch processed, to ensure there is no double-counting. (In order to read the entire output from the beginning, simply change the `group.id` to a different string.)

In [ ]:
```
conf = {'bootstrap.servers': broker_name, 'group.id': "foo1",
        'default.topic.config': {'auto.offset.reset': 'smallest'}}

c = Consumer(conf)

c.list_topics().topics[request_id]
c.subscribe([request_id])
timeout = 10.0  # Need a long timeout to allow for partition assignment
running = True
all_mass_hists = None
```

In [12]:
```
n_total_events = 0
while running:
    msg = c.poll(timeout=timeout)
    if msg is None:
        running = False
        continue
    if msg.error():
        raise KafkaException(msg.error())
    else:
        # Proper message
        buf = msg.value()
        reader = pa.ipc.open_stream(buf)
        batches = [b for b in reader]
        for batch in batches:
            arrays = awkward.fromarrow(batch)
            v_particles = uproot_methods.TLorentzVectorArray.from_ptetaphi(
                arrays['Electrons_pt'], arrays['Electrons_eta'],
                arrays['Electrons_phi'], arrays['Electrons_e']
            )
            v_particles = v_particles[v_particles.counts >= 2]
            diparticles = v_particles[:, 0] + v_particles[:, 1]

            mass_hist = hist.Hist('Counts', hist.Bin('mass', r'$m_{\mu\mu}$ (GeV)', 150, 0.0, 150.0))
            mass_hist.fill(mass=diparticles.mass/1000.0)

            if all_mass_hists:
                all_mass_hists = all_mass_hists.add(mass_hist)
            else:
                all_mass_hists = mass_hist

            n_total_events += len(arrays.tolist())
            # print("Number of events: " + str(n_events))
```

The output above is aggregated into a single histogram for plotting:

In [13]:
```
print(n_total_events)
fig, ax, _ = hist.plot1d(all_mass_hists)
plt.show()
```

```
1993800
```

Open source technologies used